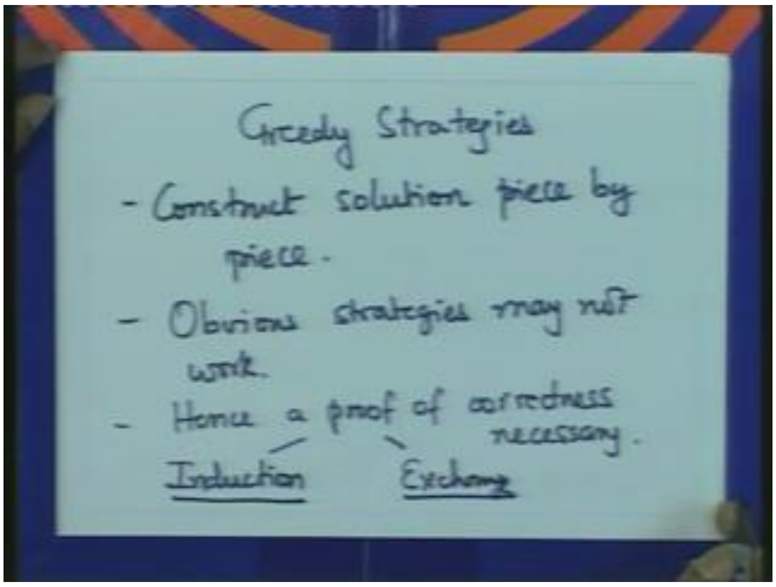
Design and Analysis of Algorithms
Prof. Sunder Vishwanathan
Department of Computer Science Engineering
Indian Institute of Technology, Bombay

Lecture – 11
Greedy Algorithms – II

We are discussing Greedy Strategies for Algorithm Design. The main principle is to somehow construct solution to your problem piece by piece. We saw two examples, last time. Let us quickly short of revise the main lessons that will learn.

(Refer Slide Time: 01:18)



Greedy Strategies

- Construct solution piece by piece.
- Obvious strategies may not work.
- Hence a proof of correctness necessary.

Induction Exchange

So, greedy strategies for algorithm design, has been reviewed. Main point is, you somehow construct the solution, piece by piece. So, two things are important. One is, how you, what is the next piece of your solution. That is the crucial question that you have to answer. And once, you answer this question, you just put it around in the loop and you get your algorithm.

The other thing we solve, especially with the 2nd problem, 2nd problem was to find the, input was a set of intervals. And you are the output was, maximum number of intervals that do not overlap with each other. Pair wise, they do not overlap with each other. Now, we saw that two obvious strategies for this problem field. One was to pick, smallest interval or the strategy of picking and interval which overlapped, smallest number of other intervals, both of these seen to be natural strategies to consider in this field.

The strategy that worked before last time was to pick first the interval that ends first. To pick an interval that ends first, remove the intervals at overlap with this. And when you request on the rest, again you pick an interval that ends first and so on. So, the set of moral of that story was obvious strategies may not work. What is obvious to one may not be obvious to the other. But, certain obvious strategies, which appear obvious to most normal human beings, may turn out not to work.

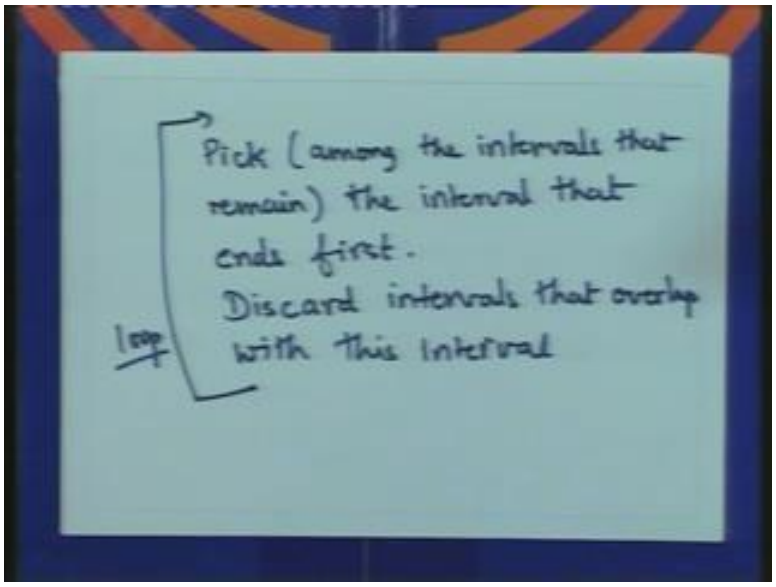
So, the point that I want to stress is a proof of correctness is important. You come up with the greedy strategy; you need to argue logically, that this strategy works. So, let me write system, this is important. So, hence a proof of correctness is necessary and important. Because, obvious things do not work, if you are giving in obvious solution to your problem, then you go to say that proof that this obvious solution is indeed correct. So, this is important.

There are usually two ways of writing a proof for these greedy strategies. One is induction. Since, you are constructing the solutions, piece by piece. You find the first piece, argue that this is correct. And then you proof of correctness, usually works by induction on the rest of the input. So, you argue that on the rest of the input, your strategy by induction works.

And added to the first thing that you pick, it gives your optimal solution. This is, what we followed yesterday. Now, this one more way of writing this, which is also quite popular, which I will call exchange. So, this is proof by the exchange techniques. This is very similar to the exchange trick that I have been mentioning throughout. And we look at it the bit more detail.

What we will do is, take up this 2nd problem that you look at. An actually, write the proof derive a proof by using this exchange business, just you get gathered into ((Refer Time: 05:41)). So, here is of the proof by exchange words.

(Refer Slide Time: 05:51)

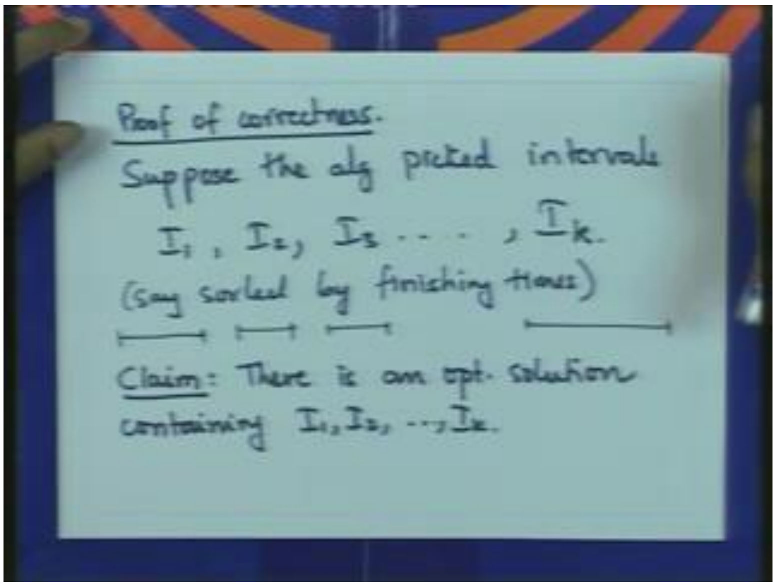


Pick (among the intervals that remain) the interval that ends first.
Discard intervals that overlap with this interval

loop

Let us review the algorithm first. It is says, pick, this is among the intervals that remain, the interval that ends first. Discard intervals that overlap with this interval. The whole thing is in a loop. This whole thing is in a loop and you are done, when if discarded all the intervals. We have an empty set. Start out with the set of intervals, do this in a loop till you do not have intervals to deal with. Now, this was the algorithm, we know that, this works. Let us now proof, using this sort of exchange strict that disworks. The proof is actually logically the same, only it is written the bit differently.

(Refer Slide Time: 07:18)



Proof of correctness.

Suppose the alg picked intervals

$I_1, I_2, I_3 \ldots, I_k$.

(Say sorted by finishing times)

Claim: There is an opt. solution containing $I_1, I_2, \ldots, I_k$.

How does this work, suppose, so here the proof of correctness. It is the 2nd proof of correctness. Suppose, the algorithm picked intervals of say I 1, I 2, I 3 so on to I k. Let us also, suppose that, they are sorted, say sorted by does not matter, whether it is finishing times or finishing times. Whether, it is sorted by finishing times, they are also sorted by starting time, because these intervals are disjoint. So, the intervals, we look like this.

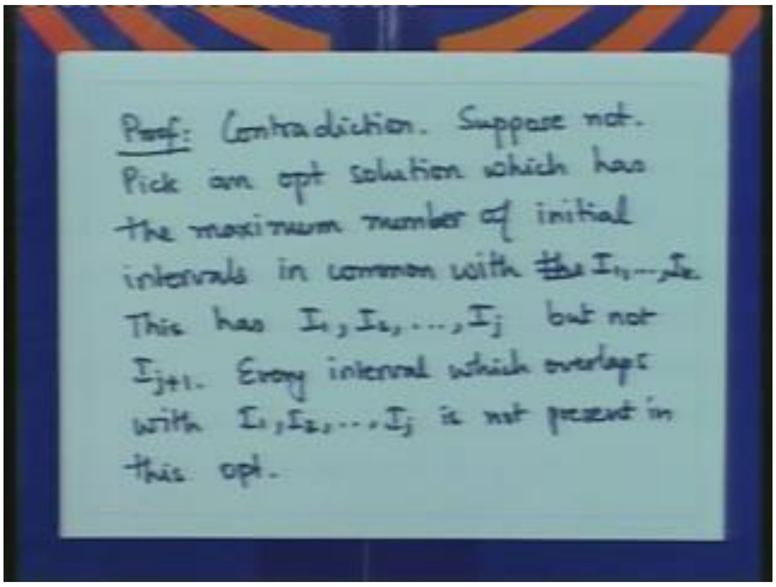
So, this is I 1, it is I 2, I 3, so on, next I k. So, the intervals are disjoint. So, sorting by finishing times and starting times is the same thing. Suppose, the algorithm, pick these intervals I 1, I 2, I 3 up to I k. Now, we have to somehow of show, that there is an optimum solution, which also picks these intervals. If this is lead to, take the optimum, if this is an optimum solution, which as I 1, I 2, I 3 up to I k, then you are done.

Because, every other interval in the input, must overlapped with at least one of these inputs. The way the algorithm works, it picks an interval, throughout the intervals, which overlap with this. And finally, we have an empty set. So, there is a set of intervals, that we picked as I 1 through I k, it is set. The rest of it has been discarded. Because, they have been, they overlapped with one of these intervals.

So, if any optimum solution has I 1, I 2 to I k in it, then this is it, your optimum must be must have size k. So, let us pick. So, this is what we need to show that, you know optimum, this is an optimum solution. Suppose, it is not so, there is no optimum, which as I 1, I 2, I 3 up to I k. Then, pick the optimum solution, which has maximum number of initial intervals in common with for the algorithm pick.

Let me write this term. So, suppose, so the claim that you want to make is, there is an optimum solution containing I 1, I 2 and so on up to I k. Well, in this case, we just saw that, if contains these, it must be exactly this, which case this is optimum. So, if you prove this claim, we have done.

(Refer Slide Time: 10:44)



Proof: Contradiction. Suppose not. Pick an opt solution which has the maximum number of initial intervals in common with ~~the~~ $I_1, \ldots, I_k$. This has $I_1, I_2, \ldots, I_j$ but not I_{j+1}. Every interval which overlaps with $I_1, I_2, \ldots, I_j$ is not present in this opt.

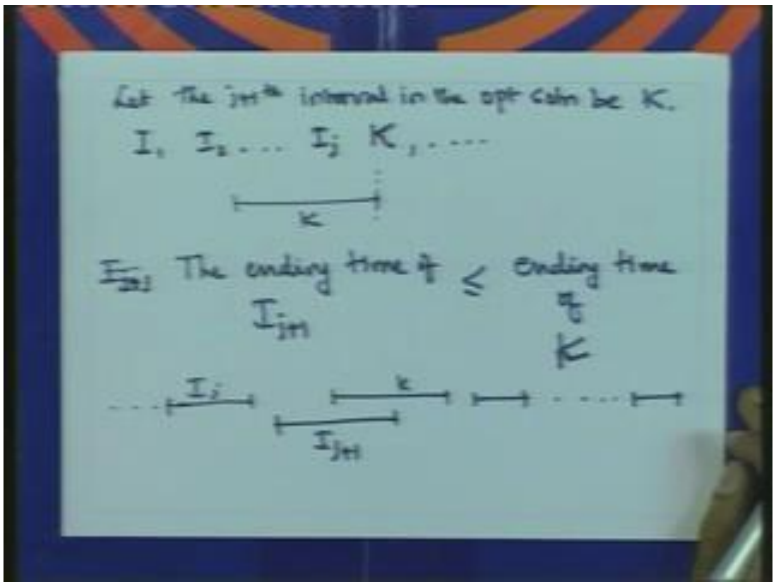
So, the proof is, why by contradiction, proof of the claim by contradiction. So, we start of by saying, suppose not, which means; suppose the statement is false, there is no optimum solution, which contains I 1, I 2 all the way up to I k. That is, why this is. So, then, pick an optimum solution, which has the maximum number of initial solutions, initial intervals in common with I 1 up to I k.

What do I mean, so there maybe some optimum solutions, which contain I 1, which do not contain I 1. There maybe some, which contain I 1 and I 2, but not I 3. There may be some with contain I 1, I 2, I 3 all the way up to I 17, but not I 18 and so on. So, all these optimum solutions pick 1. So, that the maximum number of initial intervals or in this optimum solution.

So, suppose then, this optimum solution has I 1, I 2, I 3 up to I j. So, this has I 1, I 2 up to I j, but not I j plus 1. So, this says that, there is an optimum solution, which has I 1, I 2 up to I j and no optimum solution has I j plus 1. When, I look at all these optimum solutions, there could be many, none of them has I j plus 1 and then. But, there is one, which as I 1, I 2 up to I j. So, let us look at, just picks this and look at it.

Now, since it has I 1 through I j, every interval, which overlaps with these is not present in the optimum. So, every interval, which overlaps with I 1, I 2 up to I j is not in the optimum present in this optimum. In fact, it is not present, even in the algorithms. So, let us look at I j plus 1.

(Refer Slide Time: 14:20)



Let the $j+1^{th}$ interval in the opt soln be K.

$I_1 \; I_2 \dots \; I_j \; K, \dots$

I_{j+1} The ending time of I_{j+1} $\leq$ ending time of K

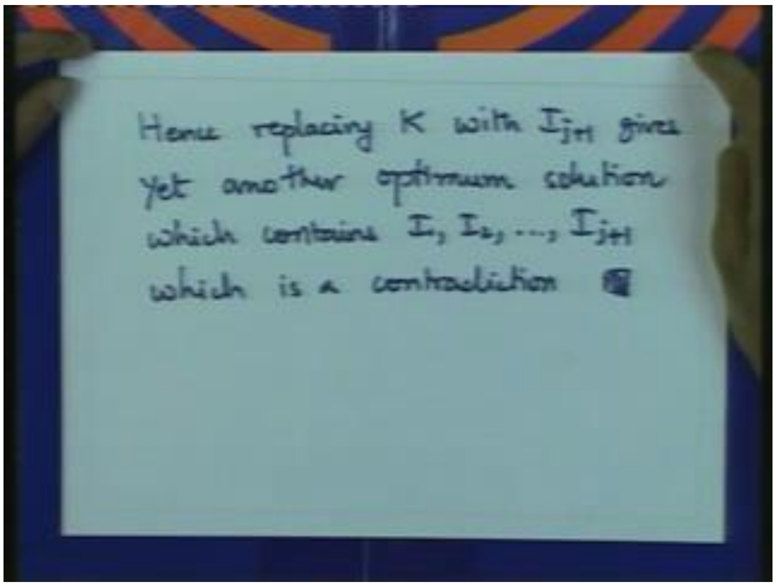
So, here the optimum, it has I 1, I 2 up to I j. Then, it has some other interval, Kolid interval k, j plus 1th interval in the optimum is k. So, let the j plus 1th interval in the opt solution be k. So, it is look like this and then, there are others. This is of the optimum looks. Now, I j plus, so k looks like this, this is k. Now, the claim is that, I j plus 1, ends before the ending time of k. So, this is the crucial, let me write this down.

The ending time of I j plus 1, this is what the algorithm picked. It is less than equal to ending time of this interval k. Any time of I j plus 1 is less than equal to ending time of k. Well, which means I j plus 1, it must look like this. I have k, this is k and I j plus 1 should look like this. It could be the same as k. But, it is suddenly less than equal to this and I j sits like this. This one, this is the picture that we have.

Now, we notice that, I can put I j plus 1 into this optimum and remove k. So, look at this picture. So, in this picture, your optimum has other intervals k and so on. This is what the optimum looks like. I can insert I j plus 1, the only other interval, it intersect with this k. So, I can insert I j plus 1 and I can remove k. Because, this is the way the algorithm picked I j plus 1 and because, you pick and interval with this smallest ending time. That is the reason the ending time of I j plus 1 is smaller than equal to ending time of k.

So, I can insert I j plus 1, remove k and this is the contradiction. Because, now I have an optimum of the same size, which has I j plus 1, that is the contradiction. Let me just write the term.

(Refer Slide Time: 17:16)

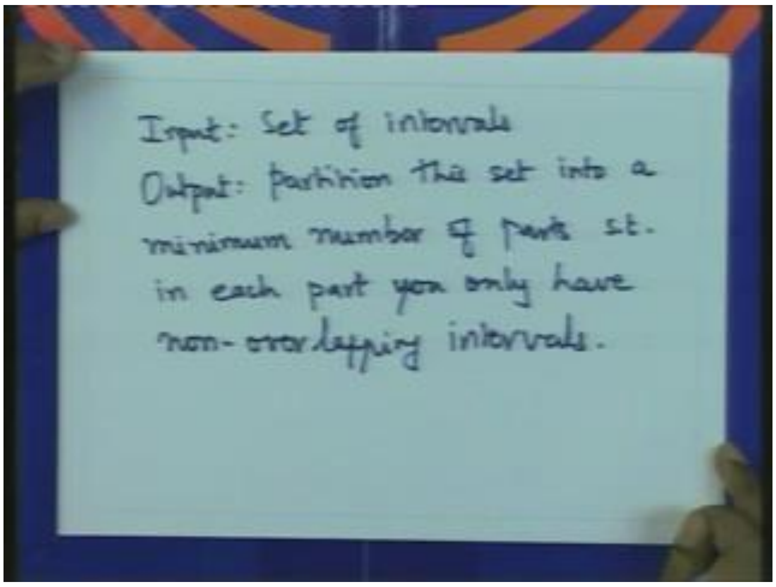


Hence replacing K with I_{j+1} gives yet another optimum solution which contains $I_1, I_2, \ldots, I_{j+1}$ which is a contradiction ∎

So, hence, replacing k with I j plus 1, gives yet another optimum solution, which contains I 1, I 2 up to I j plus 1 now, which is a contradiction; that is it. So, this is the proof by contradiction, proof by this, exchange techniques. By somehow, the exchange what the optimum has, with what the algorithm constructor. As I said earlier, you could write the same proof by induction 2. That is another way of writing the same proof and you can pick, whichever way you want to write, whichever way you like.

So, this was the previous example. Let us look at some more problems. The idea is, to give you a flavor of some kinds of algorithms are problems and algorithms. Some of them are easy and you know, you follow your notes and you get, exactly what you want. Some of them are will like this, one they may be slightly more non-trivial. So, this is just to give you a feel of behinds the things that even do with greedy algorithms.

(Refer Slide Time: 19:17)



Input: Set of intervals
Output: partition the set into a minimum number of parts s.t. in each part you only have non-overlapping intervals.

So, here is the next one, the input set of intervals as in the previous case. The output is, to partition. This set into a minimum number of parts, such that, in each part, you have, only have, non-overlapping intervals. So, this is similar, the input is similar. So, let us go back to the scheduling problem; that we had previously. Previously, you had in the set of intervals. Each interval is associated with the user.

So, the user gives an interval, where she wants to work. And your job is to sort of, you know, satisfy as many users as possible. Only, two users cannot work on this machine at the same time. So, the intervals should not overlap. Now, you still want to schedule them. Only, you have more machines, you have many machines, on the same machine, you cannot schedule to users at the same time. So, I cannot schedule to intervals, which overlapped on the same machine. And I want to minimize the number of machines.

So, users come with these requirements, which is an interval. Each user comes within interval. You have a number of machines. You want to minimize the number of machines that we use. So, which means, have to tell, for each user, which machine, he or she should sit-down. And it should be search that, for each machine, two users, whose times overlapped, should not sit on the same machine.

So, if I look at the schedules for one machine, the intervals should be non-overlapping or disjoint. So, this is the problem, then abstract way the input is the set of intervals and the output is to partition. This set of intervals into minimum number of parts. These are thing of each part is the processor. Within each part, I must only have non-overlapping intervals.

In other words, within each part, I cannot have to intervals at overlap. So, this is the problem; that we would like to solve. So, how this one, go ahead doing this, again the sort of, since we are discussing greedy techniques. I guess, one would like to do it by, you know, this keep partitioning. These intervals putting them in different parts, one by one till you are done. So, look at these intervals in some orders say and you know, this side, which part, which processor that person going to sit down.

Again, the crucial question is, which orders do you look at intervals. You can try ending time, through I mean, since the previous time, in the previous problem, you know, we looked at intervals on based on ending times. For interval at end it first was schedule on, was looked at first in the interval that in the next, we looked at. Since, so on this is the order in which we looked at the intervals. Does this work, I let you figure out in this works.

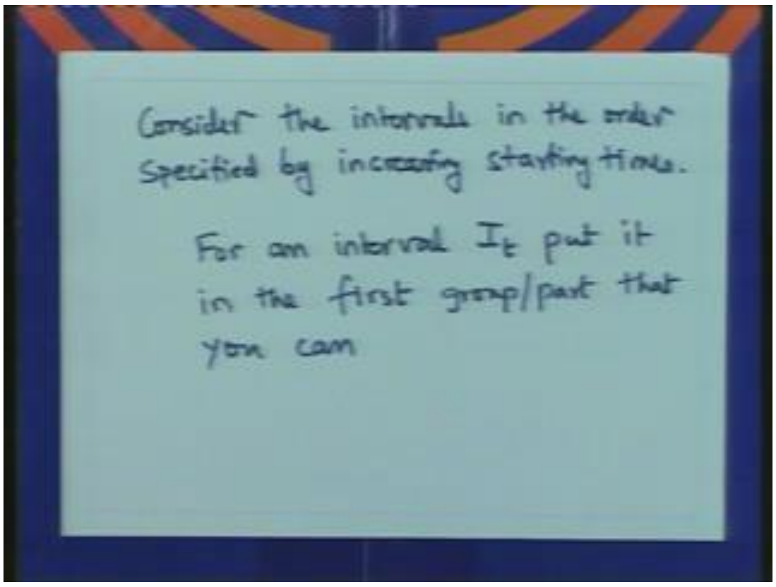
So, that is an exercise for you, just see, if you can look at intervals based on ending times, when you get a new interval, put in on one of these processors, which processor, well we have to decide that. The set of greedy way of putting, you know, putting these intervals on to processors this. Order these processors 1, 2, 3, 4, 5, 6, 7, 8, 9 and so on. And when you get an interval, put it on the first processor; that you can.

So, there is some interval on the first processor to which this overlaps. Then, I do not put it in and so on. So, I just keep going on to 1, 2, 3, 4 and the first place, where I can put it, I put it. With this sort of greedy back drop, supposing you look at intervals by finishing times, this is what, I want to check, whether this will give you an optimum solution. Well, we look at it differently will first sort by starting times.

We will sort intervals by starting times; we consider an inputs in this order. And what will do is this, when I come to the ith interval, I check, I order these groups 1, 2, 3, 4, 5. I check, whether I can put it into the first group, if I can well and good. If not, I will look at the second group and so on. So, I put it into the first group, where I can, this is the algorithm.

So, let us write this down. Then, we will see, why if at all this is optimum. They does not seem to be any, I mean on the top on set of on the phase of it. There is no real intuition as of now guiding me. Mean, why should I choose intervals, why should I order them by the starting time. I could order them by the ending time; I could have order them by sizes or any other parameter that you may have. But, well, we will do it by starting times, means see what happens.

(Refer Slide Time: 26:22)


Consider the intervals in the order specified by increasing starting times.

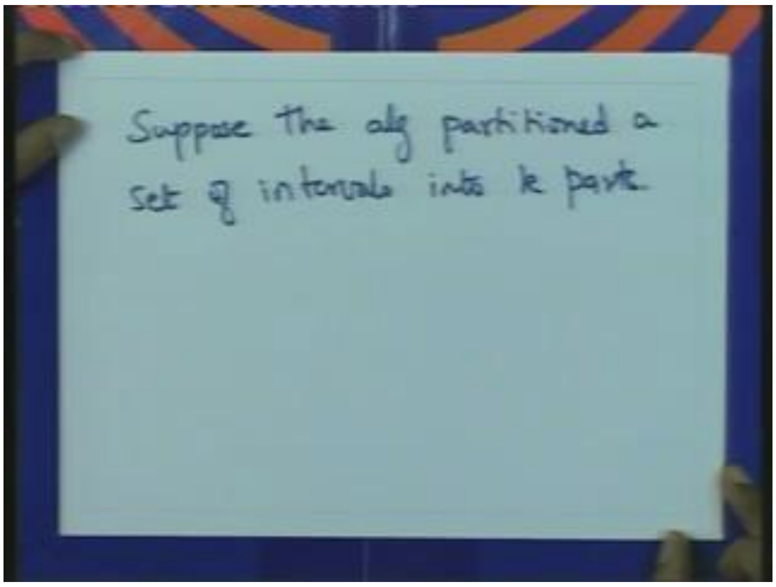
For an interval I_t put it in the first group/part that you can

So, here is the algorithm, start consider the intervals in the order specified by increasing starting times And so for an interval I, let us say I t. This is the t th interval, when considered in this order. Put it in the first; put it in the first group or part; that you can. So, you consider intervals in this order and when I look at an interval I t, put into the first parts that you can, which means in the first part in which it does not have a non, it does not have an overlapping interval.

If I put it into part L, say; that means, when I look at part 1, part 2, part 3 up to part L minus 1; in all these, there is some interval, which overlaps with I t. So, this is algorithm and why does it work, does it work. Well, it does, the algorithms does work. Now, let us in fact, prove it does works. So, well, you run the algorithms on a set of inputs. So, initially, the first interval goes into the first part. 2nd interval, if it does not overlap, if the first one goes into the first one. If it does overlap, it goes into the 2nd part and so on. So, you just sort of scan.

Your objective is some of to figure out, what happens to this to the number of parts, you are minimizing the number of parts. To supposing the number of parts working, the algorithm produced k parts; it is broken up into k parts. Somehow, we need to argue, I mean, if it is in fact optimum, that any algorithm will produce these k parts.

(Refer Slide Time: 29:28)



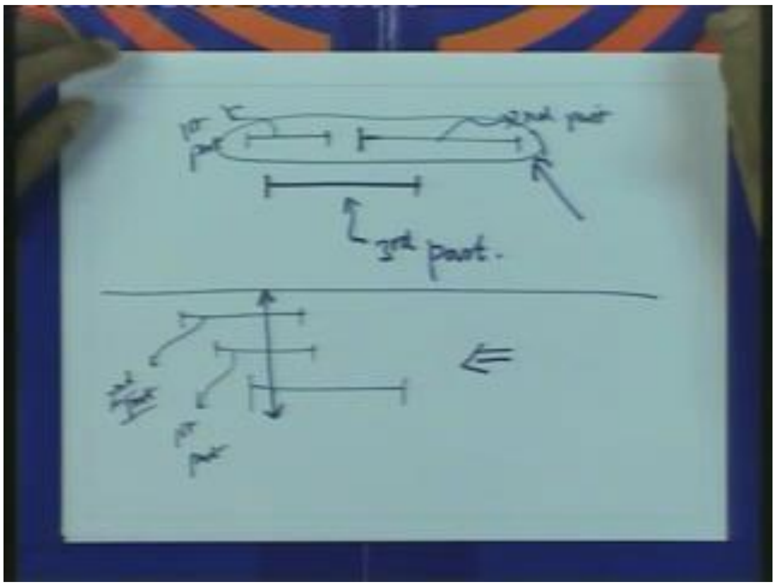
Suppose the alg partitioned a set of intervals into k parts

So, let us see why? So, let us say suppose the algorithm partitioned a set of intervals into k parts. Somehow, we have to now argue; that any algorithm on this ends set of inputs would use k parts. Why should this be true? Well, let us looks at small values of k. So, often, the other sort of trick, which is never sort of mention in text books is you should always trial small values of these parameters, you see, what really happens.

For instance, if the algorithm took two parts, if it divided into two parts, then you can you can see that, even the optimum will take two parts. Why is that so? Where, look at anything that landed in the 2nd part. Why was this interval put into the second parts? Because, it overlaps with something different parts. Now, if you have two overlapping intervals, you better you two parts.

So, if you two parts, you are through, anybody else in this you know working with the same input will use two parts. What about three parts? Well, what do you know; you know that, if something was put into the 3rd part. It must overlap with some interval in the first part and some interval in the 2nd part. Let us look at this picture.

(Refer Slide Time: 33:48)



So, here let us say I have an interval. So, this was put into the 3rd part. This interval was put into the 3rd part, which means, there must be an interval that overlapped, which was the first part and second part. Now, what are the various things that can happen, then I can have, this was in the first part. And let us say, this was in the 2nd part. This is one seen. So, what is the other seen in that possible. Let us say, this was in the first part and let us say, this was in the second part.

Let us look at these 2. Now, in this scenario, I claimed that, any other algorithm will also use three parts, why is that? Well, at this line here, when I look at this line here, all three of them intersects. So, all three of them must be put in different locations in different parts. So, if it is this situation, then we are done, we have doing optimally as usual. Let us look at this situation. Now, in this situation, we cannot see the same thing. And optimum could have up put these two into one part in which case we have ((Refer Time: 33:08)).

Well, can this situation happen, elsewhere, we see, you may progress, because remember we started by, we sorted these by starting times. So, it looks like this cannot happen. This 2nd scenario cannot happen. So, focus on this. So, in this scenario, it looks like this fellow, starts after this, which means it should be consider afterwards. So, this scenario cannot happen.

And in fact, our ordering, when we order, the ordering thing is to in fact, get read of these. So, I guess, you know, how will an algorithm design or go about doing this. Well, you say, let me look at in some order, intervals are looked at in some order. Now, look for a proof, when you look for a proof, you see, how you have to tune your algorithm. So, that the proof works and that is how the algorithm develops.

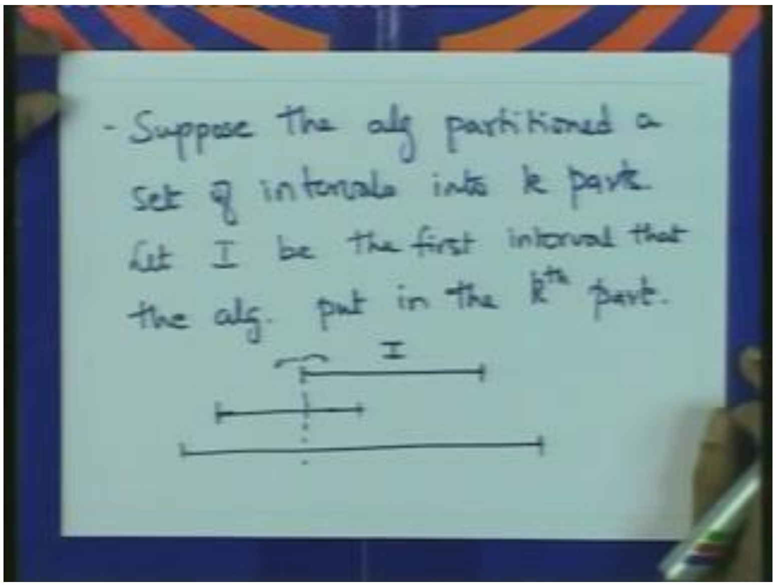
So, here, we want to look at these intervals in some order and put them into parts, put them into the first parts that it can. We see that, if there is a point, where all of them intersect, when you are through, the only way, this cannot happen is, you know, if you have a figure like that. So, have an interval and have two intervals at do not intersect into different parts.

But, if this happens, one of them must have started after this 3rd interval. One of them must start, after this interval that I am looking at right now. Then, you go back and see, can I get read out this will, can I get read out this in my proof. To get read of this, now you notice that, if you had looked at these intervals in increasing order of starting times, this case will not arise.

And so, in rank to prove, you are actually refining your algorithm, you see that, where you are previous algorithms can go wrong. And you refine this algorithm to take care of this case and you know, hopefully come up with this right algorithm. Before, looking at the problem, where you have a set of intervals and you want to partition, this set into minimum number of parts. So, that each part, you have non-overlapping intervals, if this is the problem.

And the algorithm, we will like to consider is sort in intervals by starting time, look at them in increasing order of sort in staring time. And when you look at an interval, put it into the first part; that you can. The first interval goes into the first part is 2nd interval will go into the 2nd part, if it overlaps with the first and so on. So, this is the algorithm and in fact, you would like to prove that, this algorithm is optimum. So, to prove this, well we say supposing the algorithm produced k parts. We need to show that, any algorithm will produce k parts. So, let us look at this.

(Refer Slide Time: 36:40)


- Suppose the alg partitioned a set of intervals into k parts. Let I be the first interval that the alg. put in the kth part.

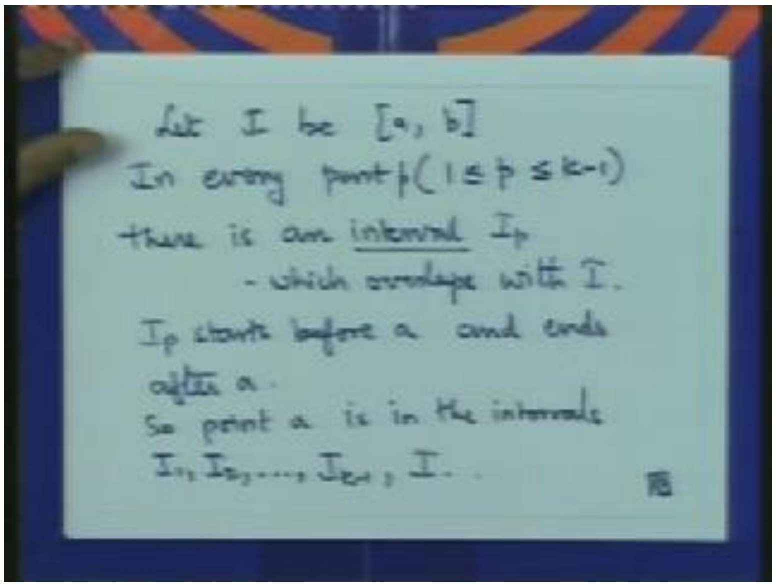
Since, there are k parts. Let I be, let us say the first interval, the rest put into the kth part. The first interval does not matter, any interval you can pick from the kth part. But, let us look at the first one, interval that the algorithm put in the kth part. We would like to now argue that, any algorithm will use k parts, why is that. So, here is I, so this my interval I. Well, if I take any other interval, it has to start before I.

Any other interval, which is already been considered. So, if I look at any other interval, that is already being consider just start before I, this much I know. I also know, when that in each of these parts. In the parts, 1, 2, 3, 4 up to k minus 1, first k minus 1 parts, there must be an interval, which intersects with I. How is that look like? So, let us say in the the first part, how can interval will be.

Well, it has to start before this and it has to be intersect this. So, it has to end after this point. So, it has be something like this. It could, where it could start here and it could end after I, that is fine. But, it has to start before this point and it should end after this point. That is the crucial thing. So, these intervals in the first part 2nd part and so on. That I overlap with, must start before this and end after this, which means, this point must be common to all of these intervals.

So, all of these intervals overlap at this point and they must be put into different parts and you have k parts. There are k intervals, which overlap at this point and hence, they must be put into k different parts. So, any algorithm will use k, k different parts, this is the ideal. So, let us write (Refer Time: 39:07)).

(Refer Slide Time: 39:09)



Let I be $[a, b]$

In every part p $(1 \leq p \leq k-1)$

there is an <u>interval</u> I_p

- which overlaps with I.

I_p starts before a and ends after a.

So point a is in the intervals

$I_1, I_2, \ldots, I_{k-1}, I$.

So, let I be the interval a b. In every part and this is one less then equal to every part P. So, one less than or equal to P, less than or equal to k minus 1, there is an interval, which overlaps with I, because I was not put into this part, which overlaps with I. Since, I was not put into any of these parts; it was put into k, part k. Now, what can be say, this means this interval, let us say I p.

So, what can we say about I p, well I p starts before a and ends after a. It is starts before a, because I consider the intervals in increasing order of starting times. And it ends after a, because it has to overlap with I. So, this point a, is in all of these intervals. So, point a, is in the intervals, I 1, I 2 up to I k minus 1 and I. So, there are k intervals, which has point a. So, we optimum must have size k. That is the end of the proof.

I am not hiding that last sentence, just fill it in, your optimum must have size k, because of this, because of k intervals, which has a point and all of them. So, the algorithm produces an optimum solution. So, that is what, we want to proof. So, well, what I want to point out with this, this problem is that in the previous case, we looked at them in increasing order of ending times.

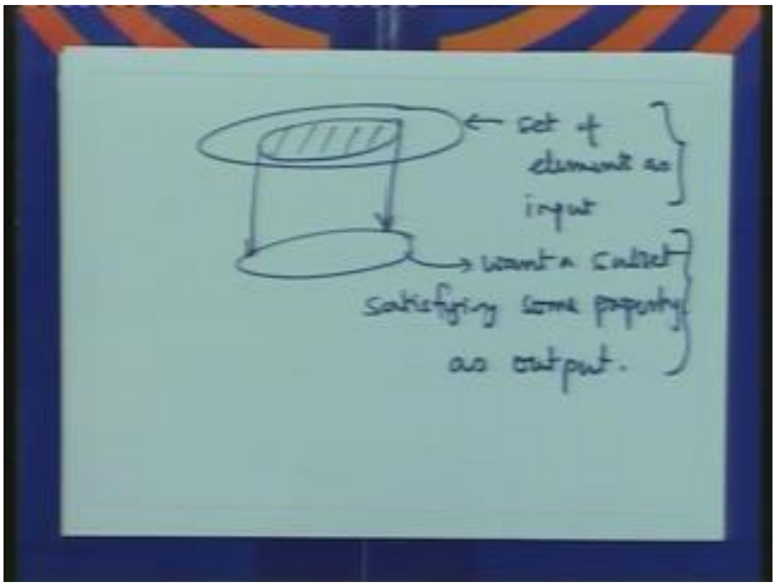
And this case, we look at them with increasing order sorting times, you could look at them with decreasing order of ending times and that will work. I will leave it you, again is an exercise. So, I can look at these intervals in decreasing order of ending times and then, do this partitioning and that will also work. Same proof, you just the whole thing is similar. I guess you can see, because I can either look at the intervals from left to right or right to left, it really does not matter.

So, things that work in one problem, you know the ordering, which works in one problem, may not work in some other problem. And this ordering of the input, the order in which you consider the input is a crucial strategy in greedy algorithms. So, remember, when I started, I said there are three set of paradigms, there are three techniques, that is run through all algorithm design, all these algorithms that we design.

One is induction. That is, we can solve it for a smaller problem, how do you extend the solution to a bigger problem. 2nd is ordering the input, look at the input in the right order. And the 3rd was storing old values, some of store value, which may be reusing. In greedy strategy, this greedy strategy that you looked at, you can see that, there are two things induction and ordering, which has so for lead the crucial role.

So, let me go back and tell you, bit more about this exchange trick. Then, we will see a problem, where it couple of problem, where this works beautifully. So, the exchange trick usually works, when your algorithm looks like this. When your problem looks like this.

(Refer Slide Time: 43:37)

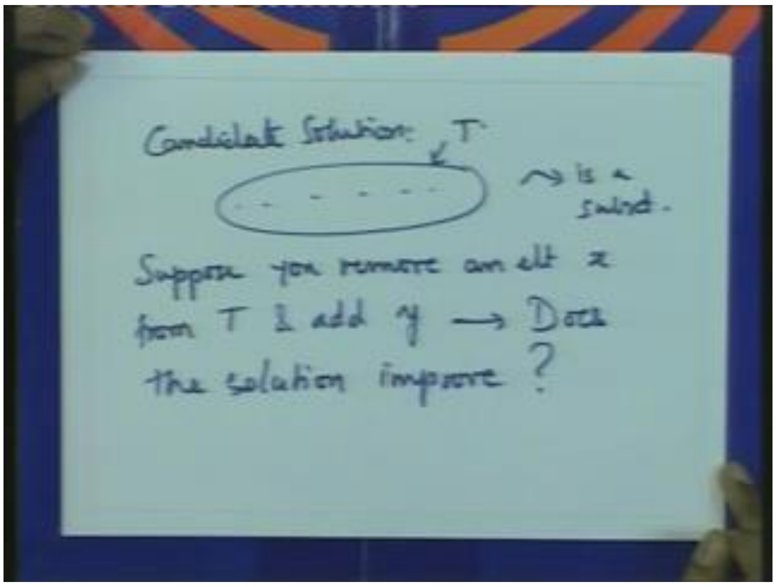

← set of element as input
→ want a subset satisfying some property as output.

So, you have a set of elements as input. You want a subset as output. And usually, you want this subset to satisfy some property. So, this is, how the problem looks like, your input is the set of elements and you want a subset of this. So, let us say, that is this, you want subset of this with satisfy some property as output. The first two problems were absolutely like this.

What was your input, initially it was a 3. It is set of vertex and well there are (Refer Time: 44:39)). There is an input with some structure, embedded in. You want a subset of the vertex set. So, that you know they are independent, you want an independent set as output, which is maximum in size. So, the first problem had this property. This 2nd problem also had this property.

You want to set of intervals, you want to subset with maximum size. So, that the intervals do not overlap. Both these problems are this property. The third one was slightly different, you wanted to partition, means and intervals into many parts. So, the first and second fitted into this roughly this frame work. So, what is this exchange trick that I am saying, roughly, it is this.

(Refer Slide Time: 45:37)



Candidate Solution: T

is a subset.

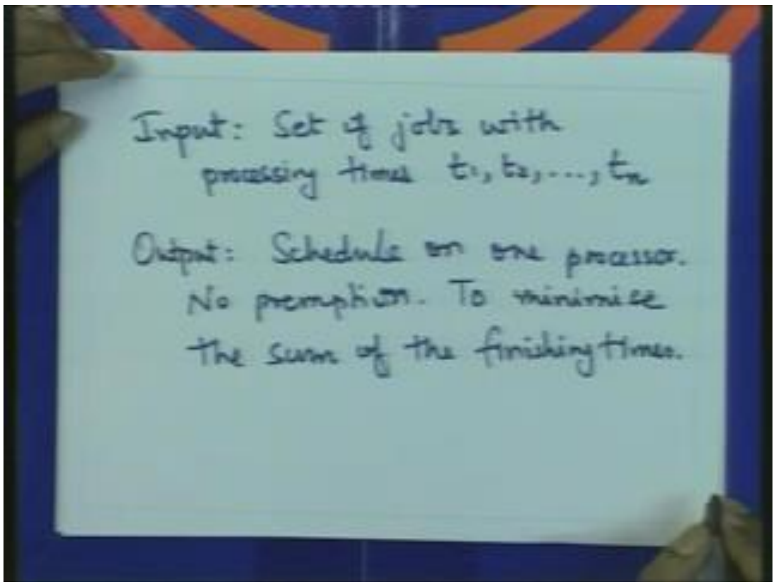
Suppose you remove an elt x from T & add y → Does the solution improve?

To supposing you have a solution, supposing you have, let us say candidate. These things are to help you design the algorithms. I am not giving you algorithm as search, but telling you tips as to, how do design this algorithm. So, your candidate algorithm solution is a subset. Now, the exchange trick is this. Supposing, you remove, let us say an element, x from this solution. So, called this T, so this is subset T from T and add, let us say y.

The question you ask you, does the solution improve, we ask this question. So, if we can some more identify, when the solution improves that, you give as a clue as to how to design these algorithm. And that happened in the first two cases. So, let me look, tell you recall your ((Refer Time: 47:05)) memory effect. So, the first case, we set that, if we did not have a leaf. So, I can put this leaf in and I remove something else, solution may improve.

In the 2nd case, I said that, you could pick, you could put, always put the interval that ended first into the solution and remove something else. That is how, we sort of came off to that. Now, we will see examples, where this was, this exchange trick was, the problem is this. So, actually fairly the simple problem and even without this exchange pick should be able to come up with algorithm.

(Refer Slide Time: 47:54)


Input: Set of jobs with processing times $t_1, t_2, \ldots, t_n$

Output: Schedule on one processor. No preemption. To minimize the sum of the finishing times.

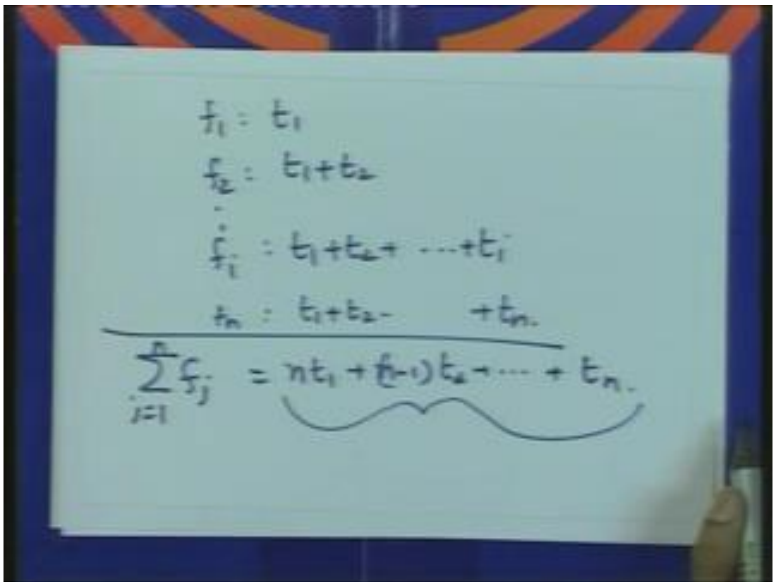
It is again from scheduling. So, you have a set of jobs. The input is set of jobs with processing times t 1, t 2, up to t n. There are n jobs, they have processing times t 1, t 2 up to t n. There is a single processor. You have one processor, you want to schedule these. So, obviously, will schedule them by one by one, there is no preemption. In fact, in all over scheduling problems, that you have taken up so for; there is no preemption.

So, you cannot stop a job in the middle, you lock at the processor. The processor runs to completion. So, you want to schedule this, one processor, no preemption. So, the output in the schedule on one processor, it is no preemption. Well and we beyond want to minimize something, want to we want to minimize, we want to minimize the some of the finishing times. Schedule of one processor, no preemption, minimize the sum of the finishing times.

So, what do I am mean by this, in fact, what do I am mean by schedule. The schedule means, as to tell you, which order you put these jobs on to the processor. So, supposing I put them in order t 1, t 2, t 3 up to t n, this is the order in which I schedule them on to the processor. The finishing time on the first job is t 1. The finishing time of the 2nd job is t 1 plus t 2. The finishing time of the 3rd job is t 1 plus t 2 plus t 3.

The ith job is, t 1 plus t 2 up to t I. This is the finishing time of the ith job. I want to minimize the sum of the finishing times. So, each job as a finishing time, I add up all finishing times. And I want to minimize the some of the finishing times. This is the problem. So, if I just look at this sum, what is this? So just take a minute.

(Refer Slide Time: 51:13)



$$f_1 : t_1$$
$$f_2 : t_1+t_2$$
$$\vdots$$
$$f_i : t_1+t_2+\cdots+t_i$$
$$f_n : t_1+t_2- \quad +t_n.$$
$$\sum_{j=1}^{n} f_j = \underbrace{nt_1+(n-1)t_2+\cdots+t_n}.$$

So, the first finishing time f 1 is t 1, f 2 is t 1 plus t 2, so on, f i, the ith finishing time t 1 plus t 2 plus 2 t i. Now, what is sigma f i, f j, let us say j equal to 1 to n. This is what I want to minimize. Well, it is t 1 plus t 1 plus t 2 and so on up to the last 1, f n is t 1 plus t 2 and so on plus t n. It is this some, if you sort of look at this some, it is nothing but n times t 1 plus n minus 1 times t 2 and so on up to plus t n is 1.

And when you look at this sum, immediately realize that I better have this modules job here, sitting here, t 1 should be minimum among the t's, t 2 should be the max minimum and so on, t n is the maximum time. It is a just look at this, you see that is what, that is the way should be and in fact, the proof is also obvious. And that is the algorithm. So, you just sort of the jobs by increasing times. And that is what; it should be, just by looking at that is a very simple algorithm for this problem. We will look at these problem, again just to sort of pushing in a point, when we start next time. And then we look at you know slightly more complicated problems from the greedy perspective.